Developer Connection

Search
Advanced Search

Log In | Not a Member?

Contact ADC

ADC Home > Internet Technologies > Web Content >

2-8-2002 not PRIOR ART

Remote Scripting with IFRAME

As web sites become more and more like traditional applications, the call-response-reload model used in HTTP transactions becomes increasingly cumbersome. Instead of delivering a single dynamic page, the DHTML or JavaScript developer must create a series of separate pages. The flow of the application is interrupted by page reloads whenever the client communicates with the server. Remote scripting provides a solution to this problem, easing development of complex JavaScript applications, and providing a better experience for the end user.

What is Remote Scripting?

Remote Scripting is the process by which a client-side application running in the browser and a server-side application can exchange data without reloading the page. Remote scripting allows you to create complex DHTML interfaces which interact seamlessly with your server.

If you're not clear on exactly what this means, think of the ever-present JavaScript image swap (you've coded one of those, haven't you?). In an image swap, your client-side code requests new data from the server to be displayed on the web page; in this case the request made to the server is for a new image with which you wish to replace an existing image. But what if you could ask the server for something other than an image? What if you could request a block of text? And what if your request could be more than a simple call for data? What if you could submit form data back to the server, have the server process that data and respond with an appropriate message? Of course, all of these things are already possible by relying on page reloads, but remote scripting allows complex interaction with the server that appears as seamless to the user as a simple image swap.

Remote scripting is a form of RPC (Remote Procedure Call), which is a general term used to describe the exchange of data between remote computer systems. Remote scripting opens up a great number of opportunities for the developer. Imagine a news article that can load side bars and graphical information related to the article directly into the web page when the site's visitors request it. Imagine a gallery of photo thumbnails that turn into full-sized images when clicked. Imagine a full-featured, browser-based content management system interface that allows a site administrator to edit web site copy in situ. The possibilities are limited only by the creativity of the developer.

Setting up Remote Scripting

As I'm sure you've gathered, remote scripting is a client/server technology that relies on two distinct components: the server, which is your web server, and the client, which is the JavaScript application in your web page. In remote scripting, you'll be making an RPC from your JavaScript application to your server. You'll need to be versed in both client and server web technologies to get started with remote scripting.

The client-side component is a part of the web page that you want to enable with remote scripting capabilities. It needs to be able to make a call back to the server, usually done via HTTP, and to then be able to receive and deal with the server response, if any. The server component must be ready to receive requests from the client, process the request, and then respond, returning data to the client if needed. The server component can be a script served up by your web server, such as a .php, .asp, or .cgi file, or it can be a dedicated software package that handles only RPC calls, or it can even be a simple database comprised of static files.

As you can see, the requirements for each of these components will vary greatly depending on how you choose to implement your remote scripting system.

Remote Scripting Technology Options

Remote scripting is actually a fairly broad term, and does not necessarily imply the use of any particular technology. Several remote scripting options are available to the web developer:

Java Applet/ActiveX control/Flash By embedding an ActiveX component, a Java applet, or a Flash movie into your web page, all of which are capable of making HTTP requests and interacting with your client-side

JavaScript code, you can implement the client component of a remote scripting system. Not all browsers support the technology (called LiveConnect) that allows for interaction between JavaScript and such objects; most notably, IE5 Mac does not support it at all, and NS6 (on all platforms) has such splotchy support it cannot really be called support at all. In addition to those problems, using an embedded object for remote scripting requires the end-user to install additional proprietary software. ActiveX does not work on the Macintosh platform, and Java can cause some difficulties with some versions of Internet Explorer. On the upside, Java and ActiveX can create socket connections with the server, which allows the server to interact with your application even when communication is not initiated by the client component. Unless you're developing in an environment where browser homogeneity can be assumed, these technologies may not be a good choice for remote scripting.

XML-RPC is in many ways the superior choice for remote scripting. It uses XML for encapsulating data, HTTP for its transport mechanism, and can be implemented in a wide variety of platform and software environments. XML-RPS is an open, standards-based technology that is quickly becoming the de-facto method for RPC of all kinds. The down side is that it takes a bit more know-how and work to get it up and running, requiring the installation of XML-RPC libraries on your server and in your client-side code. It also does not work with IE5 on the Macintosh.

Simple HTTP via a hidden IFRAME Using an IFRAME in conjunction with a script on your web server or a database of static HTML files, is by far the easiest of the remote scripting options available. The IFRAME has been part of the HTML specification since version 4 and is supported in nearly all modern browsers. On the server, you can use your scripting language of choice to process page requests made to the IFRAME. In the remainder of this article you'll see how to implement these components.

Using an IFRAME for Remote Scripting

Enough already with the talk! Let's get to the code. I'm going to show you everything you need to know to set up a remote scripting system with an IFRAME. I'll start with the basic set up, and describe some common browser problems and their solutions.

For starters, you'll need two web pages. The first should be called `client.html` (or some variation thereof, depending on the example). It will be the main working document and will contain nearly all of the scripts. The `client.html` will be making remote scripting calls using an IFRAME embedded in the document's HTML. The second page will be `server.html`. In a live server environment, you'd call it `server.php`, or `server.asp`, or whatever extension fits your server platform, but in this example we're not using a dynamic server component, so an `.html` extension is fine.

The `client.html` file will make RPC calls by passing arguments to `server.html` in the query string. The `server.html` will process the RPC, and write out a very simple page which contains script calls back to `client.html`. This will be clearer after we look at a simple example.

In `client.html` enter the following:

```
<script type="text/javascript">
function handleResponse() {
  alert('this function is called from server.html')
}
</script>

<iframe id="RSIFrame"
  name="RSIFrame"
  style="width:0px; height:0px; border: 0px"
  src="blank.html"></iframe>

<a href="server.html" target="RSIFrame">make RPC call</a>
```

As you can see, in it's simplest form IFRAME remote scripting simply points the `TARGET` attribute of a link to a hidden IFRAME. You can hide an IFRAME by setting its WIDTH, HEIGHT, and BORDER properties to 0px. Don't use `display:none`, because NS6 ignores IFRAMEs when the `display` property is set to `none`. If you were using the above code with `display:none` set for the IFRAME, the contents of `server.html` would load into the main `client.html` document, not into the IFRAME.

In `server.html` use the following HTML:

```
<script type="text/javascript">
  window.parent.handleResponse()
</script>
```

The `handleResponse()` function is found in our `client.html` page, and we simply use the `parent` property of the IFRAME document's `window` object to call it. See it in action here. Of course you notice that in this example we are not actually processing anything on the server. If you want, on your server go ahead and change your server page code to something like this (I'm using ASP as an example server scripting language):

```
<script type="text/javascript"<
window.parent.handleResponse('<%=request.servervariables("SERVER_SOFTWARE")%>')
</script>
```

And change the `handleResponse()` function in client.html to something like this:

```
<script type="text/javascript">
function handleResponse(msg) {
  alert(msg)
}
</script>
```

You should now get an alert message containing information returned from your server, the results of your first remote procedure call!

Should you want to pass data to `server.html`, you can either embed it in the query string of the URL specified in your link, or, to allow user interaction, use a form instead of a link, like so:

```
<form action="server.html" target="RSIFrame">
<input type="text" name="whatever">
</form>
```

Just like the link, you specify the IFRAME as the target of your form, use `server.html` to process the form data on the server, and send any data back with the `handleResponse()` method. More on working with forms later.

Problems and Solutions

You may have already realized that there are some serious problems with this simplest of remote scripting scenarios. Perhaps most seriously, this method renders the "back" and "reload" buttons in most browsers useless. Because the loading of `server.html` in the IFRAME is added to the browsers `history` object, hitting the reload button after you've loaded `server.html`, for instance, reloads `server.html` in the IFRAME instead of reloading client.html as would be expected.

NOTE: You may or not experience this problem, depending on a combination of factors including your browser version, server platform, HTTP headers, and browser settings. Trust me, if you simply target a link at an IFRAME, some users will have problems with their reload and back buttons.

A new function called `callToServer()` handles this problem by using the `replace()` method of the IFRAME document's `document` object. In addition, instead of creating an IFRAME element in the HTML, I'll let `callToServer()` create the IFRAME through the wonders of the DOM. Doing so alleviates all the reload and back button problems, and keeps the mark up free from unneeded tags.

Sadly, referencing the IFRAME's `document` object is no simple task, since IE5, IE5.5, IE6, and NS6 all provide different ways to access it. IE5, both on the Mac and PC, provides the simplest method: IFRAMEs in this browser show up in the `document.frames` array and have a `document` property. In IE5.5, IE6 and NS6, you can retrieve the IFRAME element object with `document.getElementByID()`, then, in NS6, use the object's contentDocument property, and in IE5.5+, use the document property of the IFRAME object's contentWindow property. That's quite a mouthful, isn't it? Thankfully, it's quite a bit simpler in the actual code:

```
var IFrameObj; // our IFrame object
function callToServer() {
  if (!document.createElement) {return true};
  var IFrameDoc;
  var URL = 'server.html';
  if (!IFrameObj && document.createElement) {
    // create the IFrame and assign a reference to the
    // object to our global variable IFrameObj.
    // this will only happen the first time
    // callToServer() is called
   try {
      var tempIFrame=document.createElement('iframe');
      tempIFrame.setAttribute('id','RSIFrame');
      tempIFrame.style.border='0px';
      tempIFrame.style.width='0px';
      tempIFrame.style.height='0px';
      IFrameObj = document.body.appendChild(tempIFrame);

      if (document.frames) {
        // this is for IE5 Mac, because it will only
        // allow access to the document object
        // of the IFrame if we access it through
        // the document.frames array
        IFrameObj = document.frames['RSIFrame'];
      }
    } catch(exception) {
      // This is for IE5 PC, which does not allow dynamic creation
      // and manipulation of an iframe object. Instead, we'll fake
      // it up by creating our own objects.
      iframeHTML='\<iframe id="RSIFrame" style="';
      iframeHTML+='border:0px;';
      iframeHTML+='width:0px;';
      iframeHTML+='height:0px;';
      iframeHTML+='"><\/iframe>';
      document.body.innerHTML+=iframeHTML;
      IFrameObj = new Object();
      IFrameObj.document = new Object();
      IFrameObj.document.location = new Object();
      IFrameObj.document.location.iframe = document.getElementById('RSIFrame');
      IFrameObj.document.location.replace = function(location) {
        this.iframe.src = location;
      }
    }
  }

  if (navigator.userAgent.indexOf('Gecko') !=-1 && !IFrameObj.contentDocument) {
    // we have to give NS6 a fraction of a second
    // to recognize the new IFrame
    setTimeout('callToServer()',10);
    return false;
  }

  if (IFrameObj.contentDocument) {
    // For NS6
    IFrameDoc = IFrameObj.contentDocument;
  } else if (IFrameObj.contentWindow) {
    // For IE5.5 and IE6
    IFrameDoc = IFrameObj.contentWindow.document;
  } else if (IFrameObj.document) {
    // For IE5
    IFrameDoc = IFrameObj.document;
  } else {
    return true;
  }
  IFrameDoc.location.replace(URL);
  return false;
}
```

With that function added to `client.html`, and the IFRAME tag removed, you now can simply add `callToServer()` to the link as an `onclick` event handler that returns `false` if the IFRAME exists and can be loaded with `server.html`, and otherwise returns `true`, allowing the link to function as normal. I've removed the TARGET attribute of the link, and changed the `href` also, pointing it to the blank page. You'll probably want to point the link to a page on your server which explains to the user that their browser does not support Advanced IFRAME Remote Scripting Technology, or better yet, which performs the same tasks as `server.html` and then redirects the user back to client.html.

```
<a onclick="return callToServer();" href="blank.html">make RPC call</a>
```

You can see this more complete code in action here.

Using a Form to Pass Data to the Server

Now that you've got the basic framework in place, it would be nice to add a form to send data to the server and receive back a useful response. You can't simply target the IFRAME with a form anymore, as you've learned, and you are limited to passing data to server.html via the query string. What you need is a function to shuffle values from a form into a properly formatted query string which can then append to the URL passed to the `replace()` method of the IFRAME's `document.location` property.

```
function buildQueryString(theFormName) {
  theForm = document.forms[theFormName];
  var qs = ''
  for (e=0;e<theForm.elements.length;e++) {
    if (theForm.elements[e].name!='') {
      qs+=(qs=='')?'?':'&'
      qs+=theForm.elements[e].name+'='+escape(theForm.elements[e].value)
      }
    }
  return qs
}
```

The function `buildQueryString()` takes one parameter, the name of the form from which you wish to grab data. It iterates through each element of that form, and for those elements with `name` properties, it adds a new name/value pair to a string called `qs`. The function then returns `qs`. With `buildQueryString()` in place, you can now change `callToServer()` to also accept a form name as parameter, and then within `callToServer()` we change the line

```
var URL = 'server.html';
```

to

```
var URL = 'server.html' + buildQueryString(theFormName);
```

Next, create a form like so:

```
<form name="emailForm" id="emailForm"
  action="server.html"
  onsubmit="return callToServer(this.name)">
Your name:<br>
<input type="text" name="realname"><br>
Your message:<br>
<textarea name="message" cols="50" rows="10"></textarea><br><br>
<input type="submit" value="submit">
</form>
```

Now we have a form that, when submitted, sends its data to the server through a hidden IFRAME. Your server can act on that data however you wish, interacting with client.html by calling `handleResponse()`. Here's an example that sends form data back to the server, hides the form, and displays a success message.

Looking at the code, you'll see that I changed the `handleResponse()` function in the client page so that instead of popping up an alert message it now takes care of hiding and showing the relevant page elements.

A Final Example

To help you see the possibilities that Remote Scripting affords you the developer, I've worked up one final example that loads data from a database of static HTML files.

Select a state from this menu:

Once a state has been selected, the multiple select list below will be populated with a partial list of ZIP code names drawn from a database of html files that are loaded via a hidden IFRAME. (ZIP code data drawn from sources on the US Census Bureau site).

Selecting a state from the drop down menu invokes the `callToServer()` function, which, instead of calling a single script on the server, loads into our hidden IFRAME one of 51 HTML files which contain ZIP code data for each of the 50 states (plus DC). The `onload` event of these files calls the `handleResponse()` function, which parses out the data in the loaded file and inserts new option elements into a multiple SELECT list. This example also resides here, where you can study the code in detail.

That should give you a taste for what can be done with RPC. Once you start playing around with it on your server, you'll really get a feel for the power in your hands. Try making calls to your server to retrieve data from a database, passing the data back to your client page with a call to the `handleResponse()` function, and then adding that data to your client page using DOM methods like those detailed in these Internet Developer articles: DOM-2 Part 1 and DOM-2 Part 2. Be sure to use your new skills for good and not evil!

Further Reading and Outside Resources

If you'd like to explore Remote Scripting further, let me recommend these sites to you:

- Brent Ashley's Remote Scripting Resources, which contains Brent's own RSLite and Javascript Remote Scripting (JSRS) tools along with links to other Remote Scripting resources.
- Virtual Cowboys' vcXMLRPC Library, a very nice implementation of XML-RPC in javascript.

Update

The code in this article, including examples, has been changed since the article first ran on 1/28/02. The original version failed in Inernet Explorer 5 for PC (IE5 mac works fine, as do IE5.5 and IE6 on the PC and NS6 and Mozilla) If you've incorporated any of the code from article into you projects, you'll want to update the `callToServer()` function as follows: look for the code listed directly below, and replace it to with the code block that follows it at the bottom of this page.

Replace this Code...

```
if (!IFrameObj && document.createElement) {
  // create the IFrame and assign a reference to the
  // object to our global variable IFrameObj.
  // this will only happen the first time
  // callToServer() is called
  var tempIFrame=document.createElement('iframe');
  tempIFrame.setAttribute('id','RSIFrame');
  tempIFrame.style.border='0px';
  tempIFrame.style.width='0px';
  tempIFrame.style.height='0px';
  IFrameObj = document.body.appendChild(tempIFrame);

  if (document.frames) {
    // this is for IE5 Mac, because it will only
    // allow access to the document object
    // of the IFrame if we access it through
    // the document.frames array
    IFrameObj = document.frames['RSIFrame'];
  }
}
```

...with this Code

```
if (!IFrameObj && document.createElement) {
  // create the IFrame and assign a reference to the
  // object to our global variable IFrameObj.
  // this will only happen the first time
  // callToServer() is called
 try {
    var tempIFrame=document.createElement('iframe');
    tempIFrame.setAttribute('id','RSIFrame');
    tempIFrame.style.border='0px';
    tempIFrame.style.width='0px';
    tempIFrame.style.height='0px';
    IFrameObj = document.body.appendChild(tempIFrame);

    if (document.frames) {
      // this is for IE5 Mac, because it will only
      // allow access to the document object
      // of the IFrame if we access it through
      // the document.frames array
      IFrameObj = document.frames['RSIFrame'];
    }
  } catch(exception) {
    // This is for IE5 PC, which does not allow dynamic creation
    // and manipulation of an iframe object. Instead, we'll fake
    // it up by creating our own objects.
    iframeHTML='\<iframe id="RSIFrame" style="';
    iframeHTML+='border:0px;';
```

```
        iframeHTML+='width:0px;';
        iframeHTML+='height:0px;';
        iframeHTML+='"><\/iframe>';
        document.body.innerHTML+=iframeHTML;
        IFrameObj = new Object();
        IFrameObj.document = new Object();
        IFrameObj.document.location = new Object();
        IFrameObj.document.location.iframe = document.getElementById('RSIFrame');
        IFrameObj.document.location.replace = function(location) {
          this.iframe.src = location;
        }
      }
    }
```